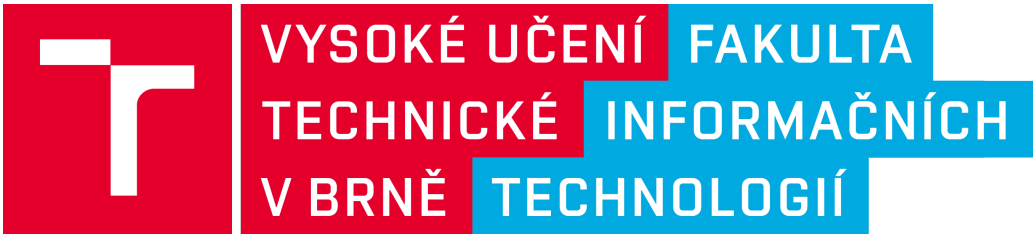


IFJ – Projektová dokumentace

Implementace překladače imperativního jazyka IFJ21

Tým 101, varianta 1

Aliaksandr Skuratovich	**(xskura01)**	30 %
Evgeny Torbin	(xtorbi00)	25 %
Lucie Svobodová	(xsvobo1x)	25 %
Jakub Kuzník	(xkuzni04)	20 %

Contents

1 Úvod

Cílem projektu byla implementace překladače imperativního jazyka IFJ21 do cílového jazyka IFJcode21. Samotný překladač je implementován v jazyce C. Jazyk IFJ21 je zjednodušenou podmnožinou jazyka Teal, který vznikl doplněním statického typování do jazyka Lua.

2 Výčet rozšíření

2.1 BOOLTHEN

V rámci rozšíření BOOLTHEN jsme implementovali podporu příkazu **if** bez části **else**, dále podporu příkazu **elseif** a typu **boolean** s hodnotami **true** a **false**. Operátory **and** a **or** jsme implementovali tak, že mají nejmenší prioritu ze všech operátorů, přičemž **or** má navíc menší prioritu než **and**. Oba tyto operátory mají levou asociativitu. Operátor **not** má stejnou prioritu i asociativitu jako operátor délky řetězce (#).

2.2 CYCLES

Rozšíření CYCLES umožňuje podporu cyklu **repeat-until** a numerické varianty cyklu **for**. Implementovali jsme dále příkaz **break** podle zadání.

2.3 FUNEXP

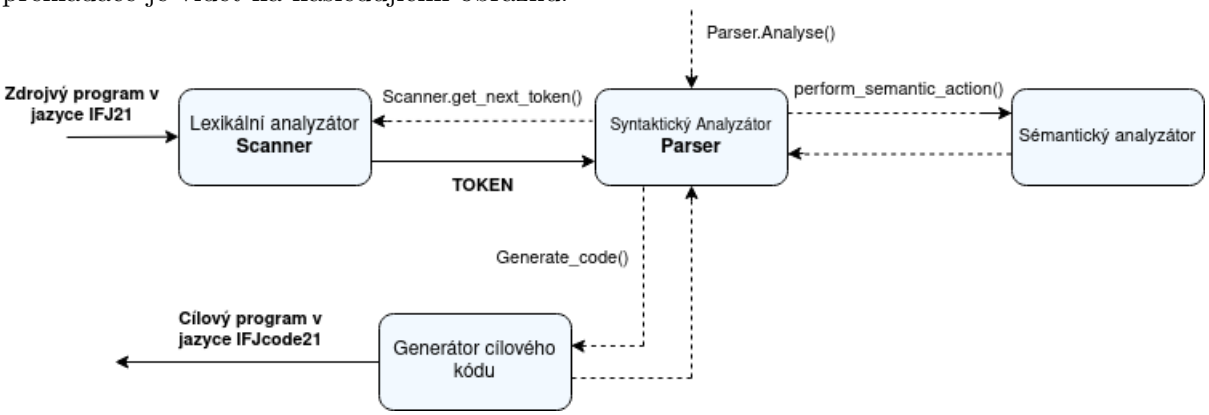
V rámci tohoto rozšíření jsme implementovali možnost volání funkcí uvnitř aritmetických, řetězcových a relačních výrazů. Kvůli tomu oproti základnímu zadání vracíme v určitých případech jiné návratové kódy. V příkazu přiřazení, při kterém nesouhlasí datové typy proměnných na levé straně datovým typům výrazů na straně pravé, vracíme vždy návratový kód 4, ať už je na pravé straně přiřazení volání funkcí, výrazy bez volání funkcí nebo obojí. Pokud je na levé straně v příkazu přiřazení více proměnných, než je výrazů na pravé straně, vracíme vždy návratový kód 7, opět bez ohledu na to, zda je součástí výrazu napravo volání funkce nebo ne.

2.4 OPERATORS

Operátor mocninu (^) jsme naprogramovali tak, že má nejvyšší prioritu a levou asociativitu. Operátor modulo (%) má stejnou prioritu a asociativitu jako násobení a operátor unární mínus (-) má pravou asociativitu a stejnou prioritu jako ostatní unární operátory, což nám dovoluje jej používat i bez závorek. V Tealu unární mínus nefunguje úplně standardně jako ostatní unární operátory, proto jsme se rozhodli implementovat unární mínus po vzoru jazyka Lua.

3 Návrh a implementace

Projekt jsme sestavili z několika námi implementovaných dílčích částí (subsystémů), které vychází z přednášek a demonstračních cvičení předmětu IFJ. Tyto dílčí části se mohou skládat z jednoho nebo více programových modulů, které spolu úzce spolupracují a dohromady tvoří překladač. Jednotlivé části překladače a jejich chování bude detailněji popsáno v následujících kapitolách. Schéma tohoto překladače je vidět na následujícím obrázku.



3.1 Lexikální analýza

Pří tvorbě překladače jsme začali lexikální analýzou (soubory `scanner.{c,h}`). Tuto část vykonává prvek zvaný Scanner. Ten pomocí DFA vytváří ze vstupního programu tokeny.

Rozhraní `Scanner` je struktura s ukazateli na tyto funkce:

```
extern const struct scanner_interface Scanner;
struct scanner_interface {
    /* Funkce, ktera zavola lexikalni analyzator, aby vratil
     * nasledujici token */
    struct token (*get_next_token)(pfile_t *);
    /* Funkce, ktera vraci token, ktery se ulozil po volani
     * get_next_token(). */
    token_t (*get_curr_token)(void);
    /* Funkce, ktera vraci typ tokenu jako c retezec. */
    char *(*to_string)(const int);
.    /* Informace o radku a pozici znaku. */
    size_t (*get_line)(void);
    size_t (*get_charpos)(void);
    void (*init)();
    void (*free)();
};
```

Hlavní funkce jsou `get_next_token()` a `get_curr_token()`, pomocí kterych si můžeme postupně žádat Scanner o další tokeny a zpracovat je, nebo se dívat na aktuální token. K tomu, aby scanner zpracoval další token, potřebuje impuls od parseru, a to tak, že parser zavolá funkci `get_next_token()` poté, co zpracuje předchozí token (viz sekce syntaktická analýza).

3.1.1 Implementace

Před implementací scanneru bylo zapotřebí navrhnout konečný stavový automat, který by chování scanneru definoval. Scanner načítá znak po znaku vstupní soubor a pomocí diagramu FSM z obrázku A rozhoduje, jak se vstupem naloží. Každý koncový stav reprezentuje, že byl načten jeden token. Pokud přijde znak, který nevede do dalšího stavu, nebo pokud jsme neskončili v koncovém stavu, vstupní program je chybný a překlad se nezdaří. Pro tyto účely používáme TOKEN_DEAD.

V jazyce C je pro přehlednost náš FSM implementován jako hlavní switch statement, ve kterém se volají funkce pro zpracování jednotlivých lexémů.
V implementaci automatu jsou také jinak pojmenované jednotlivé stavy než v samotném schématu, a to kvůli vetší přehlednosti.

3.1.2 Token

Token je reprezentován datovou strukturou, která ukládá typ tokenu a jeho atribut. Atributy jsou samostatná vnořená unie, ve které můžeme uložit identifikátory, řetězcové literály nebo čísla (`uint64_t` a `double`).

```
typedef union attribute {
    dynstring_t *id;
    uint64_t num_i;
    double num_f;
} attribute_t;

typedef struct token {
    token_type_t type;
    attribute_t attribute;
} token_t;
```

3.2 Syntaktická analýza

Hlavní řídící částí našeho překladače je parser, který se nachází v souborech `parser.{c,h}`.
Pro náš překladač jsme zvolili metodu rekurzivního sestupu řízenou LL tabulkou. Parser nám rozhoduje o tom, co budou v jakou chvíli dělat všechny ostatní subsystémy. Základní princip je ten, že si od scanneru žádá token a podle typu tokenu s pomocí gramatických pravidel rozhodne, které pravidlo použije (obrázek B). V případě syntaktické správnosti kódu může parser volat funkce sémantické analýzy. Je-li kód sémanticky správný, může se vygenerovat cílový kód pomocí volání generátoru kódu. Parser také může předat řízení překladu modulu `Expr`, a to v případě, že narazí na pravidlo obsahující pseudo-terminál ohraničený hranatými závorkami.

3.2.1 Implementační detaily

Jednotlivé non-terminály jsou v LL gramatice ohraničeny pomocí znaků `<>`. Za zmínku stojí způsob, jakým očekáváme terminály.

- Narazíme-li na situaci, kdy je nezbytně nutný nějaký terminál podle syntaktických pravidel jazyka, např. `<a> -> p` , použijeme makro `EXPECTED(p)`, které ověří typ tokenu, který získal od funkce `Scanner.get_curr_token()`. V případě když se typ shoduje s očekávaným, zavolá funkci `Scanner.get_next_token()`.
- Zatímco v případě derivace `<a> -> b <other_rules> | c` je použito makro `EXPECTED_OPT(c)`.

Parser také zajišťuje ukládání identifikátorů do tabulky symbolů, viz kapitola Tabulka symbolů.

3.2.2 Analýza výrazů

Pro analýzu výrazů používáme modul `Expr`. Ten má svoje vlastní pravidla na zpracování výrazů. Modul výraz zpracovává tak, že vkládá operátory, konstanty a identifikátory na zásobník a postupně je redukuje pomocí precedenční tabulky, která definuje, který operátor má větší prioritu. Vzhledem k tomu, že naše precedenční tabulka byla poněkud rozsáhlá, zvolili jsme implementaci pomocí precedenčních funkcí. Ke generování precedenčních funkcí jsme si vytvořili skript v jazyce Python, který z precedenční tabulky spočítá hodnoty precedenčních funkcí. Díky implementaci pomocí těchto funkcí jsme později jednoduše mohli přidávat podporu pro další operátory.

Závorky "`()`" a volání funkce "`foo()`" zpracováváme samostatně. Jelikož pro generování kódu a pro matematické výrazy používáme zásobník, kód pro každý výraz se generuje hned po jeho redukci. Stejným způsobem se zpracovávají parametry u volání funkce, to však při implementaci pomoci precedenční tabulky tvořilo hodně různých nejasností. Proto jsme se rozhodli rozlišení závorek a volání funkcí nezahrnovat do precedenční tabulky a místo toho je zpracováváme podobně jako volání funkcí, a to metodou rekurzivního sestupu.
V průběhu zpracování výrazů se provádí sémantické kontroly a po nich je generován výsledný mezikód.

Rozhraní modulu `Expr` vypadá následovně

```
1  extern const struct expr_interface_t Expr;
2  struct expr_interface_t {
3      /* Funkce, ktera zpracuje vyrazy, ktere jdou za klicovym slovem "return" */
4      bool (*return_expressions)(pfile_t *, dynstring_t *);
5      /* Funkce, ktera zpracuje vyraz, ktery jde po = v prirazeni hodnoty lokalni
       promenne
6       * nebo vyraz, ktery se nachazi v podmince bud cyklu nebo if statementu. */
7      bool (*default_expression)(pfile_t *, dynstring_t *, type_expr_statement_t);
8      /* Funkce, ktera zpracuje vyraz uvnitr funkce, a to muze byt bud prirazeni
9       * hodnoty/hodnot promenne/promennym nebo volani nejake funkce. */
10     bool (*function_expression)(pfile_t *);
11     /* Funkce, ktera zpracuje vyraz, ktery je volanim funkce na globalni urovni. */
12     bool (*global_expression)(pfile_t *);
13 };
```

Narazí-li Parser na některý z výrazů popsaných výše, které jsou v pravidlech Parseru ohraničeny pomocí hranatých závorek, dočasně předá řízení překladu modulu `Expr`.

3.3 Sémantická analýza

Funkce, které provádějí sémantické kontroly, se nacházejí uvnitř modulu `Semantics`. Důležité je zmínit, že sémantické kontroly se odehrávají v modulech `Parser` a `Expr`.
Sémantická analýza ověřuje sémantickou správnost zdrojového programu, a to pomocí kontroly datových typů. K tomu využívá datové struktury a funkce pro sémantické kontroly, které jsou implementovány v souborech `semantics.{c,h}` a jsou dostupné prostřednictvím "rozhraní" `Semantics`. Funkce v modulu `Semantics` často přistupují do datové struktury tabulky symbolů, kde mohou zjistit informace o identifikátorech a jejich rámcích, což modulu pomůže rozhodnout o sémantické správnosti kódu.

Rozhraní modulu `Semantics` vypadá následovně:

```
extern const struct semantics_interface_t Semantics;
struct semantics_interface_t {
    /* Funkce, ktera overi, zda se zhoduji typy
    /* parametru, definice a deklarace funkce  */
    bool (*check_signatures)(func_semantics_t *);
    /* Predikaty pro funkce jazyka ifj21. */
    bool (*is_declared)(func_semantics_t *);
    bool (*is_defined)(func_semantics_t *);
    bool (*is_builtin)(func_semantics_t *);
    /* Funkce, ktere nastavi hodnoty struktury func_semantics, */
    void (*declare)(func_semantics_t *);
    void (*define)(func_semantics_t *);
    void (*builtin)(func_semantics_t *);
    /* Funkce, ktere pridaji jednotlive typy parametru nebo navratovych hodnot. */
    void (*add_return)(func_info_t *, int);
    void (*add_param)(func_info_t *, int);
    /* Funkce, ktere nastavi parametry nebo navratove hodnoty funkce. */
    void (*set_returns)(func_info_t *, dynstring_t *);
    void (*set_params)(func_info_t *, dynstring_t *);
    /* Destruktor a  konstruktor datove struktury func_semantics */
    void (*dtor)(func_semantics_t *);
    func_semantics_t *(*ctor)(bool, bool, bool);
    /* Funkce overi kompatibilitu typu. */
    bool (*check_signatures_compatibility)(dynstring_t *, dynstring_t *, int);
    bool (*check_type_compatibility)(const char, const char, type_recast_t *);
    void (*trunc_signature)(dynstring_t *);
    /* Pretty print funkce a konverze pro propojeni s modulem Expr. */
    char (*of_id_type)(int);
    int (*token_to_id_type)(int);
    /* Funkce pro overeni kompatibility vyrazu. */
    bool (*check_binary_compatibility)(dynstring_t *, dynstring_t *, op_list_t,
                                       dynstring_t*, type_recast_t*);
    bool (*check_unary_compatibility)(dynstring_t *, op_list_t, dynstring_t *);
    bool (*check_operand)(token_t, dynstring_t *);
}
```

Dále modul `Semantics` využívá následující datové struktury:

```
/** Informace o datovych typech a navratovych hodnotach parametru funkce. */
typedef struct func_info {
    /* Vektor s typy navratovych hodnot a parametru. */
    dynstring_t *returns;
    dynstring_t *params;
} func_info_t;

/** Informace o semantice funkce. */
typedef struct func_semantics {
    /* Datove typy uvedene v deklaraci a v definici. */
    func_info_t declaration;
```

```
12      func_info_t definition;
13      /* Predikaty pre deklarovanou, definovanou a vestavenou funkci. */
14      bool is_declared;
15      bool is_defined;
16      bool is_builtin;
17  } func_semantics_t;
```

3.4 Generování cílového kódu

Cílovým jazykem, do něhož je jazyk IFJ21 překládán, je jazyk IFJcode21. Tento mezikód je generován již v průběhu překladu a následně je vypsán na standardní výstup v případě, že překlad vstupního programu proběhl bez chyb.

3.4.1 Rozhraní pro generování kódu

Funkce vykonávající generování jsou implementovány v souborech `code_generator.{c,h}`.

Mezikód je generován v průběhu překladu, přičemž funkce generující kód jsou volány přímo v souboru `parser.c` a v souboru `expressions.c` v případě zpracování výrazů.

Pro uchování informací potřebných ke generování je vytvořena struktura `instructions`, v níž jsou uloženy 3 lineární seznamy potřebné k ukládání vygenerovaných instrukcí. Seznam `startList` je využit pro uchování instrukcí, které mají být vypsány jako první. Tento seznam začíná úvodním instrukcí `.IFJcode21`, dále obsahuje deklaraci globálních proměnných, instrukci skoku do hlavního těla programu a návěští pro ukončení programu v případě chyby. Seznam `instrListFunctions` obsahuje definice funkcí, a to jak vestavěných a pomocných (např. pro kontrolu dělení nulou), tak i uživatelem definovaných. Posledním seznamem je `mainList`, který obsahuje instrukce v hlavním těle programu. Proměnná `instrList` obsahuje ukazatel na seznam, který je právě používán k ukládání instrukcí. Mezi seznamy je přepínáno podle charakteru vstupního kódu. Ke generování je dále využita proměnná `dynstring_t *tmp_instr`, do které je načítána instrukce, pokud se skládá z více částí (např. číslo zanoření a název proměnné). Struktura `instructions` dále obsahuje informace o tom, zda se vstupní instrukce právě nacházejí v těle cyklu, ukazatel na položku seznamu v místě před začátkem cyklu (aby bylo možné deklarovat proměnné i v těle cyklu), informace o podmíněných příkazech atd.

3.4.2 Průběh generování kódu během překladu

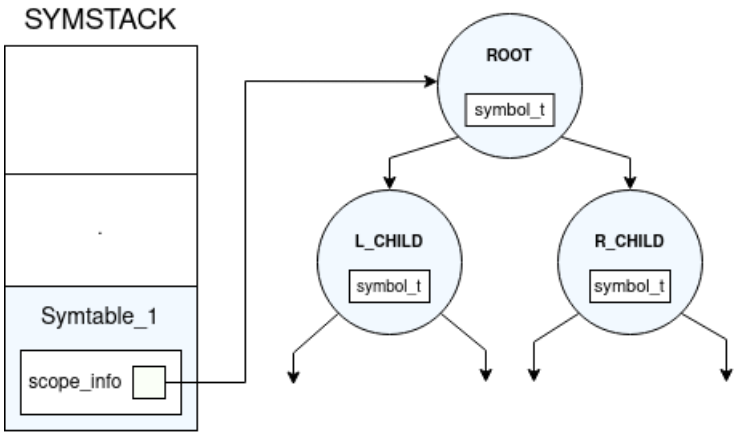
Po spuštění překladače je generátor inicializován. Následně probíhá generování kódu po syntaktické a sémantické analýze právě zpracovávaného pravidla, přičemž instrukce jsou ukládány do dříve zmíněných seznamů instrukcí. Pokud je v programu nalezena chyba, je zavolán destruktor generátoru, který uvolní alokované zdroje a výpis vygenerovaných instrukcí na standardní výstup se neprovede. V případě, že překlad proběhl bez chyb, jsou nejdříve postupně vypsány seznamy instrukcí a následně je zavolán destruktor.

Generátor kódu pro interpretaci využívá jak tříadresné instrukce, tak i zásobníkové instrukce jazyka IFJcode21. Datový zásobník je používán při zpracování výrazů. V kódu jsou použity 3 globální proměnné pro získávání mezivýsledků zpracování výrazů a pro další práci s nimi. Během zpracování výrazů jsou ošetřovány chyby dělení nulou, v případě některých operací práce s operandem typu nil, dále je prováděno přetypování `integer` → `number` a u výrazů v podmínkách i přetypování na typ `boolean`.

3.5 Datové struktury a algoritmy

3.5.1 Tabulka symbolů

Tabulka symbolů je implementována v souborech `symtable.{c,h}` jako binární strom a reprezentuje jeden rámec.



Položkou tabulky symbolů je datová struktura `symbol_t`, algoritmy použité pro práci s tabulkou symbolů jsou popsány v následujících podkapitolách.

```
typedef struct symbol {
    /* reprezentuje datovy typ a jmeno identifikatoru. */
    id_type_t type;
    dynstring_t *id;
    /* struktura obsahujici informace o navratovych typech a parametrech funkci.
     * Je inicializovana, pokud je identifikator jmenem funkce. */
    func_semantics_t *function_semantics;
    /* unikatni cislo ramce, ve kterem byl identifikator definovan.
     * Tato datova struktura je nezbytna pro generovani ciloveho kodu. */
    size_t id_of_parent_scope;
} symbol_t;
```

Následující rozhraní používáme pro práci s tabulkou symbolů.

```
extern const struct symtable_interface_t Symtable;
struct symtable_interface_t {
    /* Konstruktor a destruktor  tabulky symbolu. */
    symtable_t *(*ctor)();
    void (*dtor)(symtable_t *);
    /* Prevod typu tokenu na datovy typ. */
    id_type_t (*id_type_of_token_type)(int);
    /* Hledani a vkladani symbolu. */
    bool (*get_symbol)(symtable_t *, dynstring_t *, symbol_t **);
    symbol_t *(*put)(symtable_t *, dynstring_t *, id_type_t, size_t);
    /* Funkce prida vestavene funkce do tabulky symbolu. */
    void (*add_builtin_function)(symtable_t *, char *, char *, char *);
    /* Projde cely strom a aplikuje predikat na kazdy jeho uzel. */
    bool (*traverse)(symtable_t *, bool (*predicate)(symbol_t *));
};
```

Pro reprezentaci více rámců je v souborech `symstack.{c,h}` implementován zásobník tabulek symbolů, který poskytuje funkce pro hledání jména proměnné, jména funkce a informace o aktuálním rámci, například o typu rámce, který je potřeba ke generování kódu. Prvkem symstacku není jenom samotná tabulka symbolů, ale i další informace potřebné pro generování kódu a sémantické kontroly.

```
typedef struct stack_el {
    /* Odkaz na dalsi prvek a na tabulku symbolu. */
    struct stack_el *next;
    symtable_t *table;
    /* Jmeno funkce, pokud se jedna o ramec reprezentujici funkci. */
    dynstring_t *fun_name
    /* Typ ramce. */
    scope_info_t info;
} stack_el_t;
```

Rozhrani `Symstack` poskytuje nasledujici funkce, ktere pracuji s tabulkami symbolu.

```
extern const struct symstack_interface_t Symstack;
struct symstack_interface_t {
    symstack_t *(*init)();
    void (*dtor)(symstack_t *);
    /* Funkce, ktere volaji funkce taublky symbolu. */
    void (*push)(symstack_t *, symtable_t *, scope_type_t, char *fun_name);
    void (*pop)(symstack_t *);
    symbol_t *(*put_symbol)(symstack_t *, dynstring_t *, id_type_t);
    /* Funkce, pro hledani symbolu podle jmena promenne. */
    bool (*get_local_symbol)(symstack_t *, dynstring_t *, symbol_t **);
    bool (*get_symbol)(symstack_t *, dynstring_t *, symbol_t **);
    /* Funkce, ktera vraci tabulku symbolu na vrcholu zasobniku. */
    symtable_t *(*top)(symstack_t *);
    /* Funcke potrebne pro generovani kodu. */
    char *(*get_parent_func_name)(symstack_t *);
    symbol_t *(*get_parent_func)(symstack_t *);
    scope_info_t (*get_scope_info)(symstack_t *);
    /* Funkce, ktera aplikuje funkci-predikat pro vsechny symboly vsech tabulek symbolu
    . */
    bool (*traverse)(symstack_t *, bool (*predicate)(symbol_t *));
}
```

3.5.2 Binární strom

Binární strom jsme využili při implementaci tabulky symbolů a je implementován v rozhraní `symtable.c`. Na jednotlivé symboly tabulky symbolu nahlížíme jako na listy stromů.
Zajímavá je funkce `traverse()`, která projde metodou preorder celý strom a aplikuje funkci `predikat()` na každý jeho uzel.
Nechť x je uzel a $p(x)$ je predikát. Pak funkce `traverse()` je:

$$\bigwedge_{n \in Nodes} p(x)$$

3.5.3 Lineární seznam

Jednosměrně vázaný lineární seznam je implementován v souborech `list.c, h`. Využíváme ho k ukládání instrukcí při generování kódu a také je využit pro implementaci zásobníku.

3.5.4 Pfile

`Pfile` je datová struktura, která reprezentuje soubor programu. Tato struktura je implementována v souborech `progfile.{c,h}` a byla využívána například pro ladění, kdy jsme díky ní vytvářeli "soubory" z řetězců a ze standardního vstupu. Dále jsou k dispozici funkce `pgetc()` a `ungetc()`, které fungují stejně jako funkce z knihovny `<stdio.h>`.

4 Práce v týmu

4.1 Komunikace

Komunikace mezi členy týmu probíhala především na privátním Discord serveru určenému k projektu, kde jsme měli vytvořené kanály pro specifická témata a snažili jsme se udržovat komunikaci strukturovanou (většinou se to úplně nedařilo). Zároveň jsme měli pravidelné on-line meetingy na začátku semestru a pár osobních setkání, ta byla především určená k rozhodování o tom, co je potřeba udělat a kdo a jak to udělá, nebo pro hledání chyb v programu.

4.2 Verzovací systém

Pro paralelní práci jsme používali verzovací systém Git a vzdálený repozitář jsme měli vytvořený na serveru Github. Abychom mohli pracovat efektivně, každý člen týmu si mohl vytvořit svoji větev pro řešení části projektu, na které aktuálně pracoval. V plánu bylo větve průběžně "mergovat" do master větve pomocí "pull requests". Realita byla taková, že byl ve větvích zmatek a každý týden se vynořila nová větev, ze které se stál nový master. Ten často býval i 200 commitů pozadu za skutečnou master větví. Nicméně jsme se dozvěděli hodně věcí o tom, jak se má pracovat s verzovacími systémy. Někdy byla práce v týmu trochu složitá, především poslední dva týdny jsme měli opodstatněný pocit, že nic nestihneme, a náš vedoucí projektu požadoval přepisování některých částí kódu, nebo i celé moduly, protože měly chyby, nebo se mu nelíbil coding style, nebo byl nečitelné.
Na projektu jsme začali pracovat s začátkem semestru, ale intenzivně jsme začali implementovat až ke konci semestru viz následující obrázek z git historie.

Commit Activity

4.3 Rozdělení práce

I přesto, že měl každý člen týmu přidělenou část práce, dalo by se říct, že vždy pomáhal jiným členům s jejich částí práce.

	Scanner	Parser	Expressions	Semantics	Code Gen.	Symtable	dat. strukt.
xskura01	100%	100%	0%	40%	0%	85%	70%
xtorbi00	0%	0%	100%	60%	0%	0%	0%
xsvobo1x	0%	0%	0%	0%	100%	0%	30%
xkuzni04	0%	0%	0%	0%	0%	15%	0%

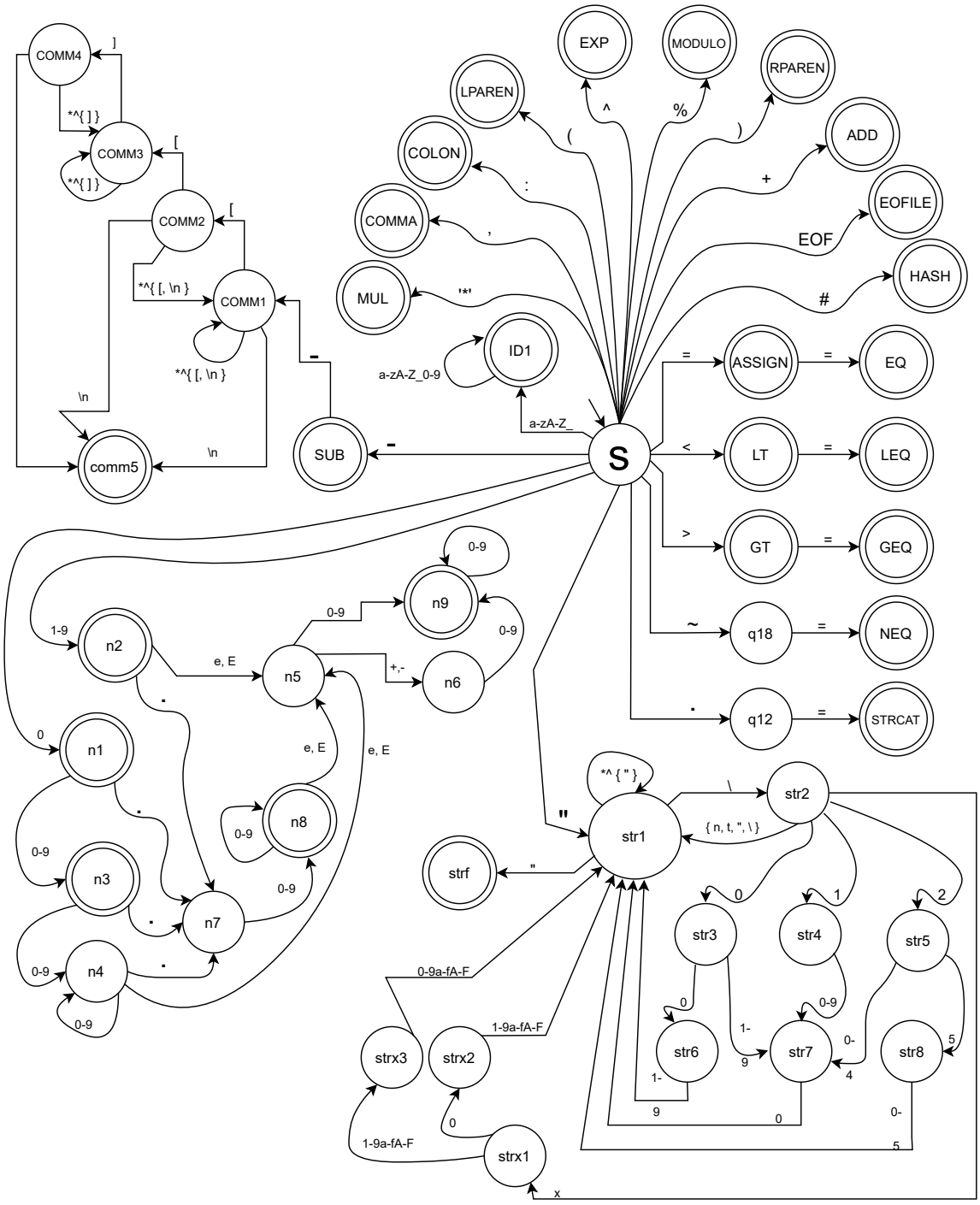
	Testování	Dokumentace
xskura01	40%	10%
xtorbi00	15%	10%
xsvobo1x	15%	10%
xkuzni04	30%	70%

5 Zdroje

[1] Přednášky a demonstrační cvičení FIT IFJ
[2] Compilers Pinciples, Techniques, Tools Second Edition Aflred V. Aho Monica S. Lam Ravi Sethi Jefrrez D Ullman
[3] https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1/

6 Přílohy

A Diagram FSM lexikální analýzy



B LL - gramatika

Hranaté závorky jsou pseudo-terminály, které znamenají, že se řízení parsingu předá analyzátoru výrazu.

1. $\langle \textbf{program} \rangle \rightarrow$ *require* "*ifj*21" $\langle \textbf{stmt\_list} \rangle$
2. $\langle \textbf{stmt\_list} \rangle \rightarrow \langle \textbf{stmt} \rangle \langle \textbf{stmt\_list} \rangle$
3. $\langle \textbf{stmt\_list} \rangle \rightarrow$ *EOF*
4. $\langle \textbf{stmt} \rangle \rightarrow$ [*global_expression*]
5. $\langle \textbf{stmt} \rangle \rightarrow \langle \textbf{function\_definition} \rangle$
6. $\langle \textbf{stmt} \rangle \rightarrow \langle \textbf{function\_declaration} \rangle$
7. $\langle \textbf{function\_definition} \rangle \rightarrow$ *function id* ($\langle \textbf{funparam\_def\_list} \rangle \langle \textbf{funretopt} \rangle \langle \textbf{fun\_body} \rangle$
8. $\langle \textbf{function\_declaration} \rangle \rightarrow$ *global id* : *function* ($\langle \textbf{datatype\_list} \rangle \langle \textbf{funretopt} \rangle$
9. $\langle \textbf{funretopt} \rangle \rightarrow e$
10. $\langle \textbf{funretopt} \rangle \rightarrow : \langle \textbf{datatype} \rangle \langle \textbf{other\_funrets} \rangle$
11. $\langle \textbf{other\_funrets} \rangle \rightarrow e$
12. $\langle \textbf{other\_funrets} \rangle \rightarrow , \langle \textbf{datatype} \rangle \langle \textbf{other\_funrets} \rangle$
13. $\langle \textbf{dataype\_list} \rangle \rightarrow \langle \textbf{datatype} \rangle \langle \textbf{other\_datatypes} \rangle$
14. $\langle \textbf{dataype\_list} \rangle \rightarrow)$
15. $\langle \textbf{other\_datatypes} \rangle \rightarrow)$
16. $\langle \textbf{other\_datatypes} \rangle \rightarrow , \langle \textbf{datatype} \rangle \langle \textbf{other\_datatypes} \rangle$
17. $\langle \textbf{funparam\_def\_list} \rangle \rightarrow)$
18. $\langle \textbf{funparam\_def\_list} \rangle \rightarrow$ *id* : $\langle \textbf{datatype} \rangle \langle \textbf{other\_funparams} \rangle$
19. $\langle \textbf{other\_funparams} \rangle \rightarrow)$
20. $\langle \textbf{other\_funparams} \rangle \rightarrow ,$ *id* : $\langle \textbf{datatype} \rangle \langle \textbf{other\_funparams} \rangle$
21. $\langle \textbf{fun\_body} \rangle \rightarrow \langle \textbf{fun\_stmt} \rangle \langle \textbf{fun\_body} \rangle$
22. $\langle \textbf{fun\_body} \rangle \rightarrow$ *end*
23. $\langle \textbf{fun\_stmt} \rangle \rightarrow$ *break*
24. $\langle \textbf{fun\_stmt} \rangle \rightarrow$ [*function_expression*]
25. $\langle \textbf{fun\_stmt} \rangle \rightarrow \langle \textbf{for\_cycle} \rangle$
26. $\langle \textbf{fun\_stmt} \rangle \rightarrow \langle \textbf{var\_definition} \rangle$
27. $\langle \textbf{fun\_stmt} \rangle \rightarrow \langle \textbf{while\_cycle} \rangle$
28. $\langle \textbf{fun\_stmt} \rangle \rightarrow \langle \textbf{repeat\_until\_cycle} \rangle$

29. ⟨***fun_stmt***⟩ → ⟨***return_stmt***⟩
30. ⟨***repeat_until_cycle***⟩ → *repeat* ⟨***repeat_body***⟩
31. ⟨***return_stmt***⟩ → *return* [*return_expressions*]
32. ⟨***while_cycle***⟩ → *while* [*default_expression*] *do* ⟨***fun_body***⟩
33. ⟨***var_definition***⟩ → *local id* : ⟨***datatype***⟩ ⟨***assignment***⟩
34. ⟨***if_stmt***⟩ → *if* ⟨***cond_stmt***⟩
35. ⟨***for_cycle***⟩ → *for id* = [*default_expression*] , [*default_expression*] ⟨***for_increment***⟩ ⟨***fun_body***⟩
36. ⟨***for_increment***⟩ → *do*
37. ⟨***for_increment***⟩ → , [*default_expression*] *do*
38. ⟨***assignment***⟩ → *e*
39. ⟨***assignment***⟩ → = [*default_expression*]
40. ⟨***repeat_body***⟩ → *until*
41. ⟨***repeat_body***⟩ → ⟨***fun_stmt***⟩ ⟨***repeat_body***⟩
42. ⟨***datatype***⟩ → *string*
43. ⟨***datatype***⟩ → *integer*
44. ⟨***datatype***⟩ → *boolean*
45. ⟨***datatype***⟩ → *number*
46. ⟨***datatype***⟩ → *nil*
47. ⟨***cond_stmt***⟩ → [*default_expression*] *then* ⟨***cond_body***⟩
48. ⟨***cond_body***⟩ → ⟨***fun_stmt***⟩ ⟨***cond_body***⟩
49. ⟨***cond_body***⟩ → ⟨***elseif_stmt***⟩
50. ⟨***cond_body***⟩ → ⟨***else_stmt***⟩
51. ⟨***cond_body***⟩ → *end*
52. ⟨***elseif_stmt***⟩ → *elseif* ⟨***cond_body***⟩
53. ⟨***else_stmt***⟩ → *else* ⟨***fun_body***⟩

C LL - tabulka

	require	glob_exp	func	glob	,	)	end	break	fun_expr	var_def	return_stmt	repeat	return	while	default_exp	do	local	if	for	=	until	string	integer	boolean	number	nil	elseif	else	$
<program>	1																												
<stmt_list>		2	2	2																									3
<stmt>		4	5	6																									
<function_definiton>																													
<function_declaration>																													
<funretopt>		9	9	8	10		9	9	9	9	9	9		9					9										9
<other_funreturns>		11	11	9	12		11	11	11	11	11	11		11					11										11
<datatype_list>				11		14																13	13	13	13	13			
<other_datatypes>					16	15																							
<funparam_def_list>					18	17																							
<other_funparams>					20	19																							
<fun_body>							22	21	21	21	21	21		21					21										
<fun_stmt>								23	24	26	29	28		27					25										
<repeat_until_cycle>												30																	
<return_stmt>													31																
<while_cycle>														32															
<var_definiton>																	33												
<if_stmt>																		34											
<for_cycle>																			35										
<for_increment>					37											36													
<assignment>																				39									
<repeat_body>								41	41	41	41	41		41					41		40								
<datatype>																						42	43	44	45	46			
<cond_stmt>															47														
<cond_body>							51	48	48	48	48	48		48					48								49	50	
<elseif_stmt>																											52		
<else_stmt>																												53	

D Precendenční tabulka

A = {*, /, //, %}
B = {+, -}
C = {<, <=, >, >=, ==, ≐}
D = {#, not, - (unary)}

	id	ˆ	A	B	C	D	..	and	or	$
id	×	>	>	>	>	>	>	>	>	>
ˆ	<	>	>	>	>	>	>	>	>	>
A	<	<	>	>	>	<	>	>	>	>
B	<	<	<	>	>	<	>	>	>	>
C	<	<	<	<	>	<	<	>	>	>
D	<	<	>	>	>	>	>	>	>	>
..	<	<	<	>	>	<	<	>	>	>
and	<	<	<	<	<	<	<	>	>	>
or	<	<	<	<	<	<	<	<	>	>
$	<	<	<	<	<	<	<	<	<	

Precendeční tabulka, kde operátor $ reprezentuje začátek a konec výrazu.

	id	ˆ	A	B	C	D	..	and	or	$
f	12	12	10	8	6	10	6	4	2	0
g	13	11	9	7	5	11	7	3	1	0

Tabulka udává nejdelší cesty terminálu k operátoru $